Realizzare un’applicazione dove sono presenti due tipologie di thread: **produttori e consumatori**

* *produttori*: inseriscono dei valori in una coda di dimensione limitata con gestione **circolare**;
* *consumatori*: prelevano i dati dal buffer.

Gestione della coda

* gli *inserimenti* sono effettuati in coda;
* i *prelievi* sono effettuati dalla testa della coda.

Vincoli

* i consumatori non possono prelevare da una coda vuota;
* i produttori non possono procedere all’inserimento se la coda è piena;
* prelievi/inserimenti sono effettuati in mutua esclusione.

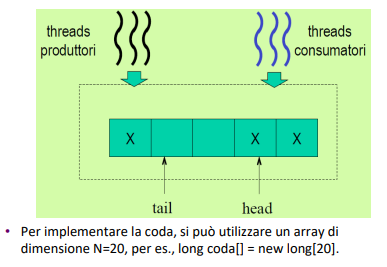


Immagine che contiene testo, diagramma, Piano, linea

Descrizione generata automaticamente

**Main**

L’esercitazione ci offre l’opportunità di scegliere diversi modi per implementare la sincronizzazione.  
In particolare nel **main** andremo a **istanziare** una coda circolare senza sincronizzazione, per poi darla come argomento ad un **Wrapper** che implementerà la sincronizzazione. Questa verrà data come parametro ai **workers**.

Creo un **array** di **worker** e successivamente istanzio i 100 worker con un **flag** che identificherà il **tipo** (Prod o Cons). Questi sono **implementati** per **ereditarietà** e il **metodo run** dipenderà dal flag.

A questo punto poi li **avviamo** tutti con lo start.

Al termine del **main** andremo a effettuare la **join**() dei thread in modo che il thread main attenda i thread figli.

Dunque, adesso descriviamo le altre classi.

**Coda** è l’interfaccia basica, in cui sono dichiarati i metodi basici implementati dalla classe **CodaCircolare**.

**CodaCircolare**   
Metodi basici.

* Costruttore che definisce la **dimensione** della coda, **numero di elementi nullo**, **istanzia** l’array che verrà poi gestito circolarmente, **definisce** **tail** e **head** pari a 0 ovviamente (indici per accedere alle posizioni in testa e coda dell’array).
* **Full**()
* **Empty**()
* **getSize**()

Poi ci sono i metodi **inserisci** e **preleva** che verranno richiamati in opportune sezioni dei codici dei **wrapper**.

* Inserimento in [**tail%size**] e vengono **incrementati** **elem** e **tail**.
* Prelevamento in [**head%size**] e viene **decrementato** **elem**  e ovviamente **incrementato** **head**.

**CodaWrapper** è l’interfaccia della **coda wrappata**, in particolare è una **classe abstract** che **implementa** l’interfaccia **Coda**. Questa interfaccia verrà implementata dai **wrapper** che implementeranno la sincronizzazione in maniera diversa.  
In questa classe sono implementati i metodi empty full e getSize che in realtà sono generali, e richiamano i metodi di **coda** che in realtà sono poi implementati nell’oggetto **CodaCircolare** che implementa l’interfaccia Coda.   
I metodi **inserisci** e **prelievo** invece saranno implementati dai **Wrapper** finali che implementeranno i meccanismi di sincronizzazione.

Ci sono poi 3 classi che implementano quest’interfaccia, ma conviene fossilizzarci solo su una.

In particolare su **CodaWrapperLock**.

**CodaWrapperLock**: semplicemente da ricordare è che va dichiarato un **Lock** istanziando una variabile **RentrantLock** che offre la possibilità di dichiare **variabili condition**, a differenza di quanto accade per il blocco synchronized ed in generale i metodi synchronized che hanno la problematica relativa al fatto che non vengono distinte le variabili condition ma c’è un un’unica variabile condition!

Vabbeh poi la sincronizzazione è identica a quanto vista in SO, dunque granularmente più fine e più comprensibile a differenza di quanto avviene coi block synchronized.

**Struttura**

**Interfaccia**: Coda semplice, implementata da CodaWrapper che è una classe abstract.

**Classi**:

* CodaImpl che implementa l’interfaccia Coda e implementa i metodi senza contare la sincronizzazione.
* CodaWrapperLock estende CodaWrapper, ciò è possibile perché è un abstract.  
  Il costruttore avrà come input proprio **Coda**, dunque CodaWrapperLock passerà al costruttore **super** questa coda in modo da richiamare il costruttore di CodaWrapper.